Лабораторна робота №15

Анотації. Рефлексія.Шаблон Observer

***Мета робот:***

- придбання навичок використання засобів анотування;

- ознайомлення з механізмом рефлексії;

- реалізація обслуговування колекції об'єктів на основі шаблону

проектування Observer;

- використання модульного тестування;

- підготовка документації на основі коментарів інструментом

автоматичної генерації javadoc.

**1 ЗАГАЛЬНЕ ЗАВДАННЯ**

Разработать иерархию классов согласно шаблону Observer и

продемонстрировать возможность обслуживания разработанной ранее

коллекции (наблюдаемый объект, Observable) различными (не менее двух) наблюдателями (Observers) – отслеживание изменений, упорядочивание, вывод,

отображение и т.д. При реализации иерархии классов использовать средства аннотирования (Annotation). Отметить особенности различных политик

удержания аннотаций (annotation retention policies). Продемонстрировать поддержку классами концепции рефлексии (Reflection).

Разработать класс для тестирования функциональности приложения.

Использовать комментарии для автоматической генерации документации средствами javadoc.

**2 ОПИС ПРОГРАМИ**

**2.1 Ієрархія та структура класів**

**class Info-організовує діалог з користувачем**

**class Calculator виконує індивідуальне завдання**

**class Serializator виконує збереження данних классу Calculator у файл**

**interface Calculator- містить основні методи калькуляторів**

**class Calculator Area –рахує площу**

**class Calculator FactoreArea – створює об єкт типу CalculatorArea**

**class Calculator Perimetr – рахує періметр індивідуального завдання**

**class Calculator FactoryPerimetr- створює об єкт типу CalculatorPerimetr**

**class GuiCalulator-додатковий класс для графічного інтерфейсу**

**class Main –викликає класс контроллер та файл з версткою**

**sample.fxml-містить верстку вікна Windows**

**class Application-організовує роботу по шаблону та точка входу у программу**

**interface Queue-робить чергу для запитів до додатку**

**class Menu-містить запити у своїй коллекції та виконує їх**

**interface Command-містить різні типи команд**

**class Command Queue –Організовує чергу команд на основі двох интерфейсів**

**2.2 Опис програми**

На рисунку 2.1 наведена структура розробленої програми

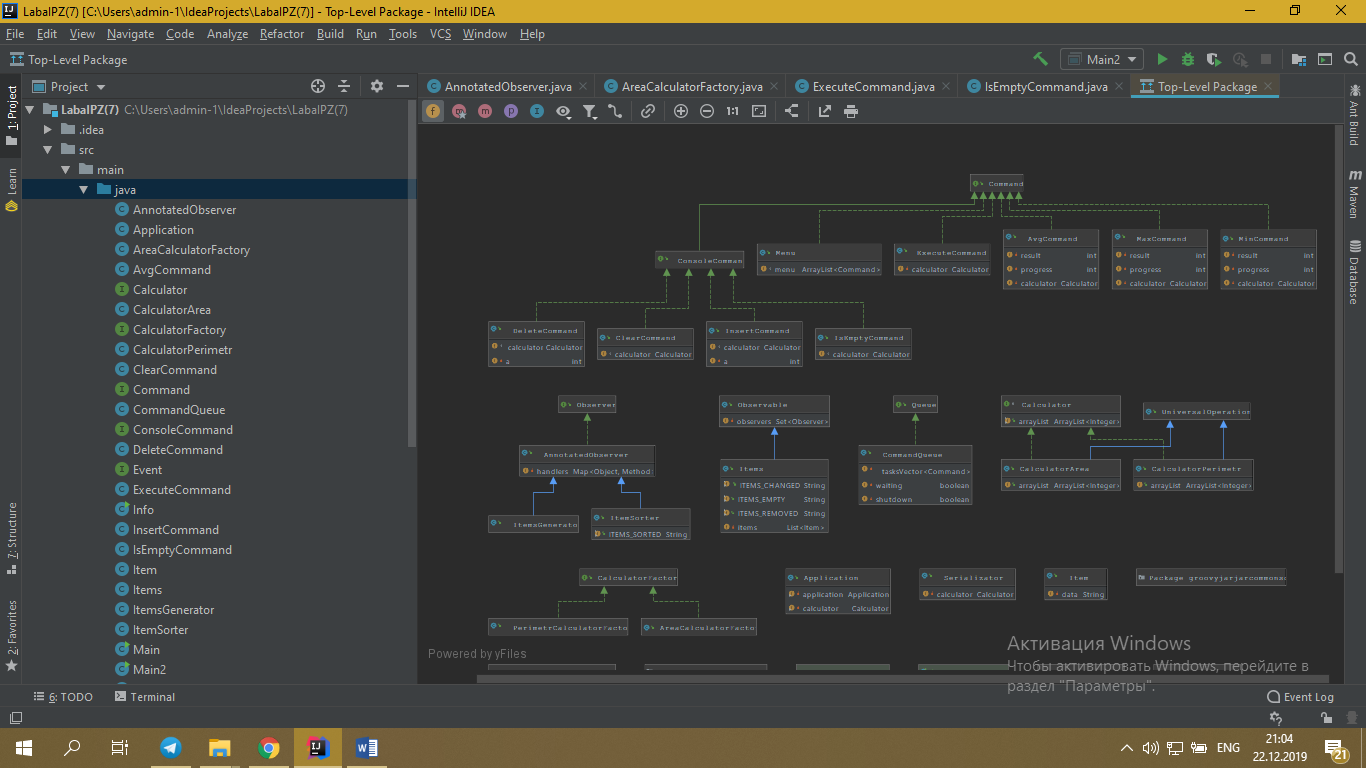


Рис 2.1 – структура програми

**2.3 Важливі фрагменти програми**

2.3.1 Файл*Observer.java*

*\* The interface Observer.  
 \*/*public interface Observer {  
 */\*\*  
 \* Вызывается наблюдаемым объектом для каждого наблюдателя; шаблон Observer  
 \*  
 \** ***@param*** *observable ссылка на наблюдаемый объект  
 \** ***@param*** *event информация о событии  
 \*/* public void handleEvent(Observable observable, Object event);  
}

2.3.1 Файл*Observable.java*

import java.util.HashSet;  
import java.util.Set;  
  
*/\*\*  
 \* The type Observable.  
 \*/*public class Observable {  
 */\*\* Множество наблюдателей; шаблон Observer  
 \** ***@see*** *Observer  
 \*/* private Set<Observer> observers = new HashSet<Observer>();  
  
 */\*\*  
 \* Добавляет наблюдателя; шаблон Observer  
 \*  
 \** ***@param*** *observer объект-наблюдатель  
 \*/* public void addObserver(Observer observer) {  
 observers.add(observer);  
 }  
  
 */\*\*  
 \* Удаляет наблюдателя; шаблон Observer  
 \*  
 \** ***@param*** *observer объект-наблюдатель  
 \*/* public void delObserver(Observer observer) {  
 observers.remove(observer);  
 }  
  
 */\*\*  
 \* Оповещает наблюдателей о событии; шаблон Observer  
 \*  
 \** ***@param*** *event информация о событии  
 \*/* public void call(Object event) {  
 for (Observer observer : observers) {  
 observer.handleEvent(this, event);  
 }  
 }  
}

2.3.1 Файл*ItemSorter.java*

import java.util.Collections;  
  
*/\*\*  
 \* The type Item sorter.  
 \*/*public class ItemSorter extends AnnotatedObserver{  
 */\*\*  
 \* The constant ITEMS\_SORTED.  
 \*/* public static final String *ITEMS\_SORTED* = "ITEMS\_SORTED";  
  
 */\*\*  
 \* Обработчик события {****@linkplain*** *Items#ITEMS\_CHANGED};  
 \* извещает наблюдателей; шаблон Observer  
 \*  
 \** ***@param*** *observable наблюдаемый объект класса {****@linkplain*** *Items}  
 \** ***@see*** *Observable  
 \*/*//@Event(Items.ITEMS\_CHANGED )  
 public void itemsChanged(Items observable) {  
 Collections.*sort*(observable.getItems());  
 observable.call(*ITEMS\_SORTED*);  
 }  
  
 */\*\*  
 \* Обработчик события {****@linkplain*** *Items}; шаблон Observer  
 \*  
 \** ***@see*** *Observable  
 \*/*// @Event(ITEMS\_SORTED)  
 public void itemsSorted() {  
 itemsSorted();  
 }  
  
 */\*\*  
 \* Обработчик события {****@linkplain*** *Items}; шаблон Observer  
 \*  
 \** ***@param*** *observable наблюдаемый объект класса {****@linkplain*** *Items}  
 \** ***@see*** *Observable  
 \*/*// @Event(ITEMS\_SORTED)  
 public void itemsSorted(Items observable) {  
 System.*out*.println(observable.getItems());  
 }  
  
 */\*\*  
 \* Обработчик события {****@linkplain*** *Items#ITEMS\_REMOVED}; шаблон Observer  
 \*  
 \** ***@param*** *observable наблюдаемый объект класса {****@linkplain*** *Items}  
 \** ***@see*** *Observable  
 \*/*// @Event(Items.ITEMS\_REMOVED)  
 public void itemsRemoved(Items observable) {  
 System.*out*.println(observable.getItems());  
 }  
}

2.3.1 Файл*Items.java*

import java.util.ArrayList;  
import java.util.Iterator;  
import java.util.List;  
  
import static jdk.nashorn.internal.objects.NativeFunction.*call*;  
  
*/\*\*  
 \* The type Items.  
 \*/*public class Items extends Observable implements Iterable<Item> {  
 */\*\*  
 \* Константа-идентификатор события, обрабатываемого наблюдателями  
 \*/* public static final String *ITEMS\_CHANGED* = "ITEMS\_CHANGED";  
 */\*\*  
 \* Константа-идентификатор события, обрабатываемого наблюдателями  
 \*/* public static final String *ITEMS\_EMPTY* = "ITEMS\_EMPTY";  
 */\*\*  
 \* Константа-идентификатор события, обрабатываемого наблюдателями  
 \*/* public static final String *ITEMS\_REMOVED* = "ITEMS\_REMOVED";  
 */\*\* Коллекция объектов класса {****@linkplain*** *Item} \*/* private List<Item> items = new ArrayList<Item>();  
  
 */\*\*  
 \* Добавляет объект в коллекцию и извещает наблюдателей  
 \*  
 \** ***@param*** *item объект класса {****@linkplain*** *Item}  
 \*/* public void add(Item item) {  
 items.add(item);  
 if (item.getData().isEmpty())  
 call(*ITEMS\_EMPTY*);  
 else call(*ITEMS\_CHANGED*);  
 }  
  
 */\*\*  
 \* Добавляет объект в коллекцию  
 \*  
 \** ***@param*** *s the s  
 \*/* public void add(String s) {  
 add(new Item(s));  
 }  
  
 */\*\*  
 \* Добавляет несколько объектов в коллекцию и извещает наблюдателей  
 \*  
 \** ***@param*** *n количество добавляемых объектов класса {****@linkplain*** *Item}  
 \*/* public void add(int n) {  
 if (n > 0) {  
 while (n-- > 0) items.add(new Item(""));  
  
 call(*ITEMS\_EMPTY*);  
 }  
 }  
  
 */\*\*  
 \* Удаляет объект из коллекции и извещает наблюдателей  
 \*  
 \** ***@param*** *item удаляемый объект  
 \*/* public void del(Item item) {  
 if (item != null) {  
 items.remove(item);  
 call(*ITEMS\_REMOVED*);  
 }  
 }  
  
 */\*\*  
 \* Удаляет объект из коллекции и извещает наблюдателей  
 \*  
 \** ***@param*** *index индекс удаляемого объекта  
 \*/* public void del(int index) {  
 if ((index >= 0) && (index < items.size())) {  
 items.remove(index);  
 call(*ITEMS\_REMOVED*);  
 }  
 }  
  
 */\*\*  
 \* Возвращает ссылку на коллекцию  
 \*  
 \** ***@return*** *ссылка на коллекцию объектов класса {****@linkplain*** *Item}  
 \*/* public List<Item> getItems() {  
 return items;  
 }  
 @Override  
 public Iterator<Item> iterator() {  
 return items.iterator();  
 }  
}

2.3.1 Файл*ItemGenerator.java*

*/\*\*  
 \* The type Items generator.  
 \*/*public class ItemsGenerator extends AnnotatedObserver{  
 */\*\*  
 \* Обработчик события {****@linkplain*** *Items#ITEMS\_EMPTY};  
 \* извещает наблюдателей; шаблон Observer  
 \*  
 \** ***@param*** *observable наблюдаемый объект класса {****@linkplain*** *Items}  
 \** ***@see*** *Observable  
 \*/*// @Event(Items.ITEMS\_EMPTY)  
 public void itemsEmpty(Items observable) {  
 for (Item item : observable) {  
 if (item.getData().isEmpty()) {  
 int len = (int)(Math.*random*() \* 10) + 1;  
  
 String data = "";  
  
 for (int n = 1; n <= len; n++) {  
  
 data += (char)((int)(Math.*random*() \* 26) + 'A');  
 }  
 item.setData(data);  
 }  
 }  
 observable.call(Items.*ITEMS\_CHANGED*);  
 }  
}

Item.java

*/\*\*  
 \* The type Item.  
 \*/*public class Item implements Comparable<Item>{  
 */\*\* Информационное поле \*/* private String data;  
  
 */\*\*  
 \* Инициализирует {****@linkplain*** *Item#data}  
 \*  
 \** ***@param*** *data значение для поля {****@linkplain*** *Item#data}  
 \*/* public Item(String data) {  
 this.data = data;  
 }  
  
 */\*\*  
 \* Устанавливает поле {****@linkplain*** *Item#data}  
 \*  
 \** ***@param*** *data значение для поля {****@linkplain*** *Item#data}  
 \** ***@return*** *значение поля {****@linkplain*** *Item#data}  
 \*/* public String setData(String data) {  
 return this.data = data;  
 }  
  
 */\*\*  
 \* Возвращает поле {****@linkplain*** *Item#data}  
 \*  
 \** ***@return*** *значение поля {****@linkplain*** *Item#data}  
 \*/* public String getData() {  
 return data;  
 }  
  
 public int compareTo(Item o) {  
 return data.compareTo(o.data);  
 }  
 @Override  
 public String toString() {  
 return data;  
 }  
}

2.3.1 Файл*AnnotatedObserver.java*

import java.lang.reflect.Method;  
import java.util.HashMap;  
import java.util.Map;  
  
*/\*\*  
 \* The type Annotated observer.  
 \*/*public class AnnotatedObserver implements Observer {  
 */\*\* Ассоциативный массив обработчиков событий; содержит пары событие-обработчик \*/* private Map<Object, Method> handlers = new HashMap<Object, Method>();  
  
 */\*\*  
 \* Заполняет {****@linkplain*** *AnnotatedObserver#handlers} ссылками на методы,  
 \* отмеченные аннотацией {****@linkplain*** *Event}  
 \*/* public AnnotatedObserver() {  
 for (Method m : this.getClass().getMethods()) {  
 if (m.isAnnotationPresent(Event.Eventt.class)) {  
 handlers.put(m.getAnnotation(Event.Eventt.class).value(), m);  
 }  
 }  
 }  
  
 public void handleEvent(Observable observable, Object event) {  
 Method m = handlers.get(event);  
  
 try {  
 if (m != null) m.invoke(this, observable);  
 } catch (Exception e) {  
 System.*err*.println(e);  
 }  
 }  
}

2.3.1 Файл*Menu.java*

import java.util.ArrayList;  
  
public class Menu implements Command {  
  
ArrayList<Command> menu =new ArrayList<Command>();  
  
  
public Command add(Command command)  
{  
 menu.add(command);  
 return command;  
}  
  
 public void execute() {  
 for (Command c:menu  
 ) {  
  
 c.execute();  
 }  
 }  
}

2.3.1 Файл*QueueCommand.java*

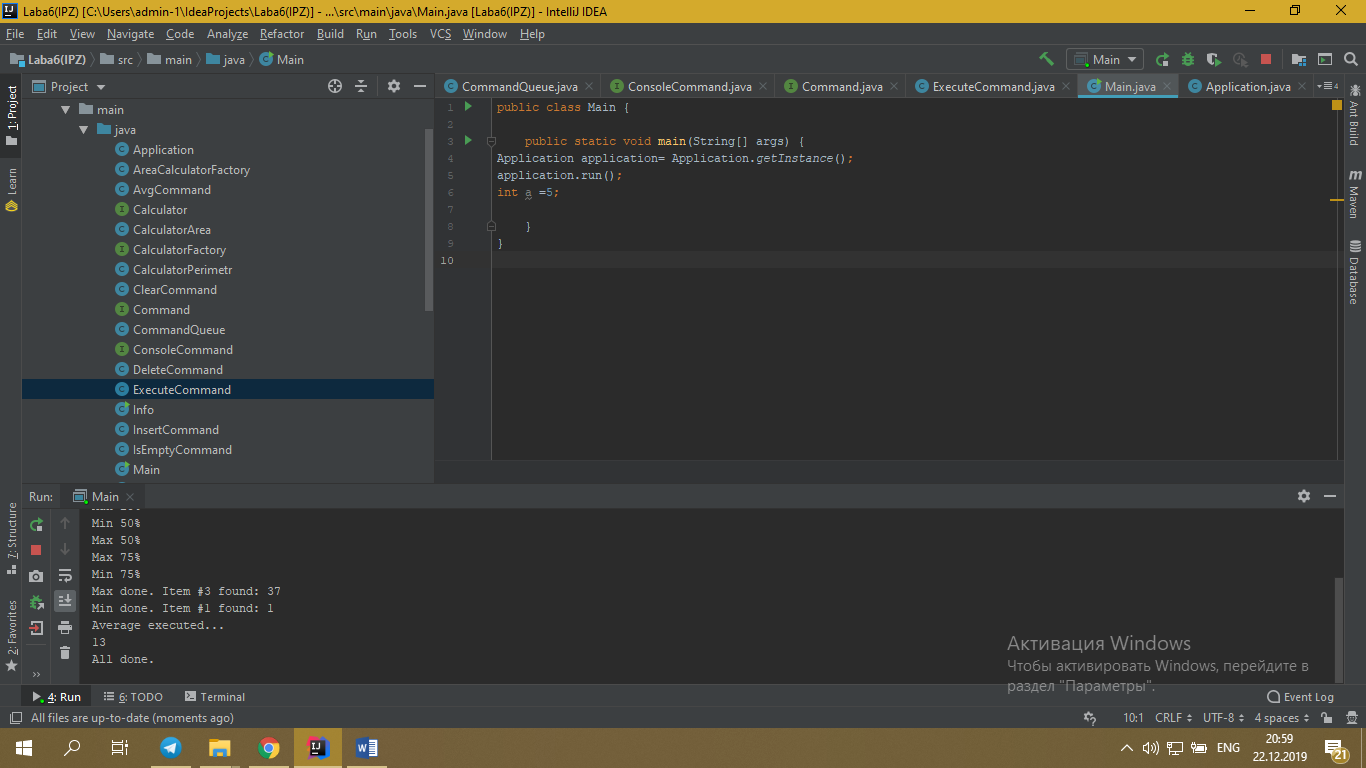
import java.util.Vector;  
  
public class CommandQueue implements Queue {  
private Vector<Command> tasks;  
  
private boolean waiting;  
  
private boolean shutdown;  
  
public void shutDown()  
  
{  
 shutdown=true;  
}  
  
public CommandQueue()  
{  
 tasks =new Vector<Command>();  
 waiting =false;  
 new Thread(new Worker()).start();  
}  
 public void put(Command cmd) {  
tasks.add(cmd);  
if (waiting)  
{  
 synchronized (this){  
 notifyAll();  
 }  
}  
 }  
  
 public Command take() {  
 if (tasks.isEmpty())  
 {  
 synchronized (this)  
 {  
 waiting=true;  
 try  
 {  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 waiting=false;  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 return (Command) tasks.remove(0);  
 }  
  
 private class Worker implements Runnable  
 {  
  
 public void run() {  
while(!shutdown)  
{  
 Command r =take();  
 r.execute();  
}  
 }  
 }  
  
}

2.3.1 Файл*ExecuteCommand.java*

public interface Command {  
  
 public void execute();  
}

import java.util.concurrent.TimeUnit;  
  
  
public class ExecuteCommand implements Command{  
  
 */\*\* Объект, реализующий интерфейс  
 \* \* обслуживает коллекцию объектов \*/* private Calculator calculator;  
 */\*\* Возвращает поле {****@linkplain*** *\* \* @return значение {****@linkplain*** *\*/* public Calculator getView() {  
 return calculator;  
 }  
 */\*\* Устанавливает поле {****@linkplaiw****}  
  
 \** ***@return*** *новое значение {****@linkpview****}  
 \*/* public Calculator setCalculator(Calculator calculator) {  
 return this.calculator = calculator;  
 }  
*/\*\* Инициализирует поле  
 \** ***@param*** *calculator объект, реализующий  
 \*/*public ExecuteCommand(Calculator calculator) {  
 this.calculator = calculator;  
}  
  
 public char getKey() {  
 return 'e';  
 }  
 @Override  
 public String toString() {  
 return "'e'xecute";  
 }  
  
 public void execute() {  
  
 CommandQueue queue1 = new CommandQueue();  
 CommandQueue queue2 = new CommandQueue();  
  
*/\*\*  
  
 ExecutorService exec1 = Executors.newSingleThreadExecutor();  
 ExecutorService exec2 = Executors.newSingleThreadExecutor();  
 /\*\*/* MaxCommand maxCommand = new MaxCommand((CalculatorPerimetr) calculator);  
 AvgCommand avgCommand = new AvgCommand((CalculatorPerimetr)calculator);  
 MinCommand minCommand = new MinCommand((CalculatorPerimetr)calculator);  
 System.*out*.println("Execute all threads...");  
  
*/\*\*  
  
 exec1.execute(minMaxCommand);  
 exec2.execute(maxCommand);  
 exec2.execute(avgCommand);  
 /\*\*/* queue1.put(minCommand);  
 queue2.put(maxCommand);  
 queue2.put(avgCommand);  
  
 /\*\*/  
 try {  
 while (  
 maxCommand.running() ||  
 minCommand.running()) {  
 TimeUnit.*MILLISECONDS*.sleep(100);  
 }  
*/\*\*  
  
 exec1.shutdown();  
 exec2.shutdown();  
 /\*\*/* queue1.shutDown();  
 queue2.shutDown();  
  
 /\*\*/  
  
 TimeUnit.*SECONDS*.sleep(1);  
 } catch (InterruptedException e) {  
 System.*err*.println(e);  
 }  
 System.*out*.println("All done.");  
 }  
}

**3 РЕЗУЛЬТАТИ**



**ВИСНОВКИ**

Навчились працювати тестами та шаблонами .